Irene Truchado Mazzoli

Pareja PT41

Arquitectura de Ordenadores 2019-2020

Explotar el potencial de las Arquitecturas Modernas

Práctica 4 – Memoria

GRUPO 1362

ÍNDICE

EJERCICIO 0

INFORMACIÓN SOBRE LA TOPOLOGÍA DEL SISTEMA

Utilizado el comando CAT /PROC/CPUINFO, obtenemos información detallada sobre nuestro equipo. En concreto, si ejecutamos dicho comando en los laboratorios de la Universidad sirviéndonos de la utilidad *grep*, obtenemos los siguientes resultados:

```
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ cat /proc/cpuinfo | grep 'cpu cores\|siblings'
siblings        : 6
cpu cores       : 6
siblings        : 6
cpu cores       : 6
siblings        : 6
cpu cores       : 6
siblings        : 6
cpu cores       : 6
siblings        : 6
cpu cores       : 6
siblings        : 6
cpu cores       : 6
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$
```

Observamos que nuestro equipo dispone de **6 *cores* físicos** (*cpu cores*) y de **6 *cores* virtuales** (*sibilings*). Es por ello por lo que podemos deducir que la opción de *hyperthreading* no está disponible, al ser el número de **cores** físicos y virtuales el mismo.

En cuanto a la **frecuencia** de dichos procesadores, volvemos a utilizar *grep* y obtenemos:

```
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ cat /proc/cpuinfo | grep 'cpu MHz'
cpu MHz         : 814.172
cpu MHz         : 801.856
cpu MHz         : 972.180
cpu MHz         : 1133.974
cpu MHz         : 817.800
cpu MHz         : 854.087
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$
```

Los resultados nos indican que cada *core* tiene una frecuencia diferente, pero todas ellas rondan entre los 800 y los 1100 MHz.

NOTA. Este es el valor de P que se utiliza los ejercicios:

P = (num_pareja mod 8) + 1

pareja PT41: P = (41 mod 8) + 1 = 1 + 1 = 2

P = 2

EJERCICIO 1

PROGRAMAS BÁSICOS DE OPENMP

En este ejercicio, vamos a realizar una primera toma de contacto con programas basados en *OpenMP*, los cuales permiten lanzar un número de hilos para que lleven a cabo tareas de forma paralela.

En primer lugar, vamos a compilar y ejecutar el programa entregado OMP1.C para comprobar cómo se lleva a cabo el lanzamiento de hilos.

Probamos lanzando 2 hilos y posteriormente 9.

```
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./omp1 2
Hay 6 cores disponibles
Me han pedido que lance 2 hilos
Hola, soy el hilo 0 de 2
Hola, soy el hilo 1 de 2
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./omp1 9
Hay 6 cores disponibles
Me han pedido que lance 9 hilos
Hola, soy el hilo 3 de 9
Hola, soy el hilo 1 de 9
Hola, soy el hilo 7 de 9
Hola, soy el hilo 8 de 9
Hola, soy el hilo 2 de 9
Hola, soy el hilo 0 de 9
Hola, soy el hilo 5 de 9
Hola, soy el hilo 4 de 9
Hola, soy el hilo 6 de 9
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$
```

1.1. Observamos que es posible lanzar más hilos que *cores* tenga el sistema. Sin embargo, esto no tiene sentido ya que en un procesador solo pueden ejecutarse simultáneamente tantos hilos como *cores* tenga la máquina. Es por ello por lo que, al exceder este umbral, los hilos comienzan a compartir recursos y a no ejecutarse de forma simultánea.

1.2. Tal y como hemos explicado, en un equipo se deberían utilizar tantos hilos como *cores* tenga dicho equipo. Si utilizáramos más, ya hemos visto que la ejecución paralela sería contraproducente, y si utilizáramos menos no estaríamos utilizando esta funcionalidad de manera eficiente. Por lo tanto, en los ordenadores de los laboratorios se deberían utilizar seis hilos. En el caso de mi ordenador personal, al disponer de dos núcleos se deberían utilizar dos hilos.

Ahora vamos a ejecutar el programa OMP2.C para estudiar las diferencias en la declaración de la privacidad de las variables de *OpenMP*. Esta es la salida del programa:

```
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./omp2
Inicio: a = 1,    b = 2,   c = 3
        &a = 0x7ffe0a8922f4,x   &b = 0x7ffe0a8922f8,    &c = 0x7ffe0a8922fc

[Hilo 0]-1: a = 0,        b = 2,   c = 3
[Hilo 0]          &a = 0x7ffe0a892290,    &b = 0x7ffe0a8922f8,    &c = 0x7ffe0a89228c
[Hilo 0]-2: a = 15,       b = 4,   c = 3
[Hilo 1]-1: a = 0,        b = 2,   c = 3
[Hilo 1]          &a = 0x7f458650ae20,    &b = 0x7ffe0a8922f8,    &c = 0x7f458650ae1c
[Hilo 1]-2: a = 21,       b = 6,   c = 3
[Hilo 3]-1: a = 0,        b = 2,   c = 3
[Hilo 3]          &a = 0x7f4585508e20,    &b = 0x7ffe0a8922f8,    &c = 0x7f4585508e1c
[Hilo 3]-2: a = 27,       b = 8,   c = 3
[Hilo 2]-1: a = 0,        b = 2,   c = 3
[Hilo 2]          &a = 0x7f4585d09e20,    &b = 0x7ffe0a8922f8,    &c = 0x7f4585d09e1c
[Hilo 2]-2: a = 33,       b = 10,          c = 3

Fin: a = 1,       b = 10,          c = 3
        &a = 0x7ffe0a8922f4,    &b = 0x7ffe0a8922f8,    &c = 0x7ffe0a8922fc

e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$
```

1.3. Al declarar una variable privada, *OpenMP* crea una copia para cada hilo paralelo lanzado. Estas copias tienen el mismo nombre para cada *thread*, pero dirección de memoria diferente. En este caso concreto, las variables privadas son A y C, y vemos que su valor en cada hilo puede variar, pero al acabar la ejecución mantienen su valor inicial, al contrario que las variables públicas, en este caso, B.

1.4. Tal y como acabamos de ver, al comenzar a ejecutarse la región paralela, el valor de una variable privada se inicializa a 0. En el caso de las variables declaradas como *firstprivate*, estas se inicializan al valor previo a entrar a la región paralela.

1.5. Al finalizar dicha región paralela, el valor de una variable privada vuelve a ser el mismo que antes de entrar en ella. Esto se debe a que los valores que hubiera podido tomar en los diferentes hilos se pierden, ya que habían utilizado posiciones de memoria diferentes a la del flujo principal.

1.6. Las variables públicas funcionan de manera diferente. Todos los hilos lanzados comparten la misma dirección de memoria, por lo que el valor de dichas variables sí se ve modificado en cada uno de los *threads* y estas terminan registrando el valor de la variable en el último hilo que finalice.

EJERCICIO 2

PARALELIZAR EL PRODUCTO ESCALAR

En este ejercicio, se va a estudiar el impacto de la paralelización del producto escalar utilizando el bucle FOR. En primer lugar, ejecutamos la versión serie y las dos versiones paralelas del producto escalar entregadas y observamos los resultados.

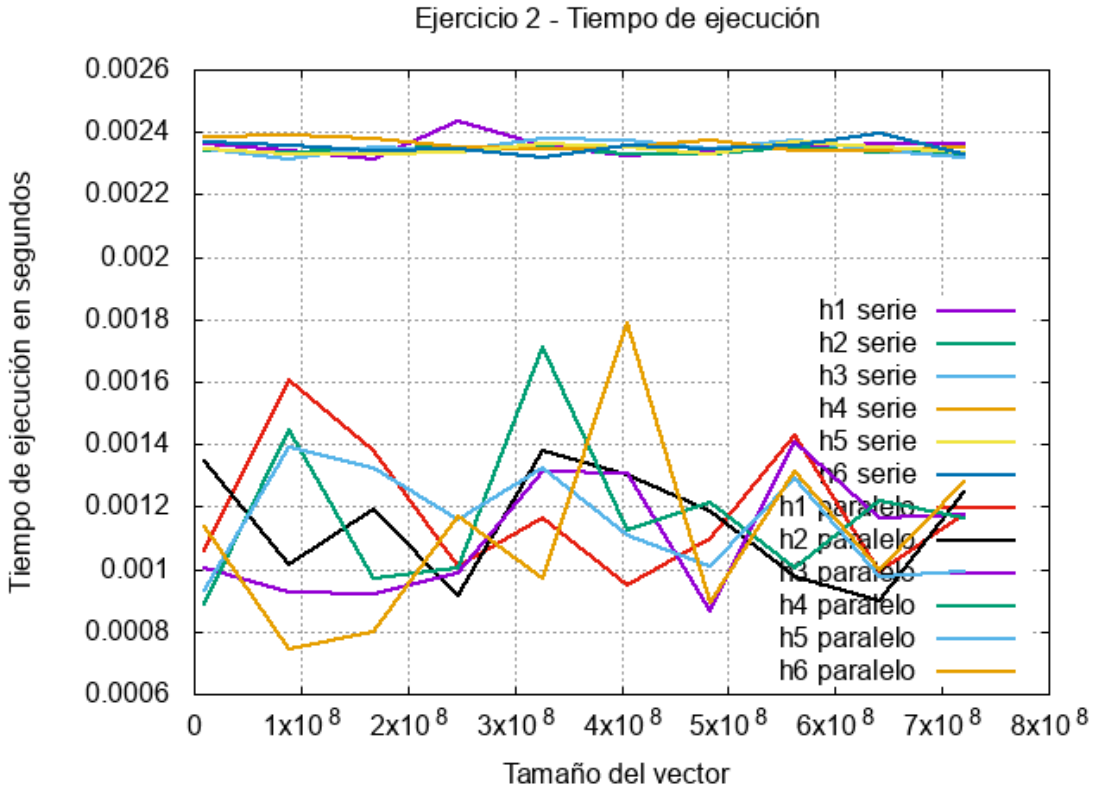
```
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./pescalar_serie
Resultado: 33.319767
Tiempo: 0.002215
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./pescalar_par1
Resultado: 7.685122
Tiempo: 0.005261
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./pescalar_par2
Resultado: 33.329933
Tiempo: 0.001252
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$
```

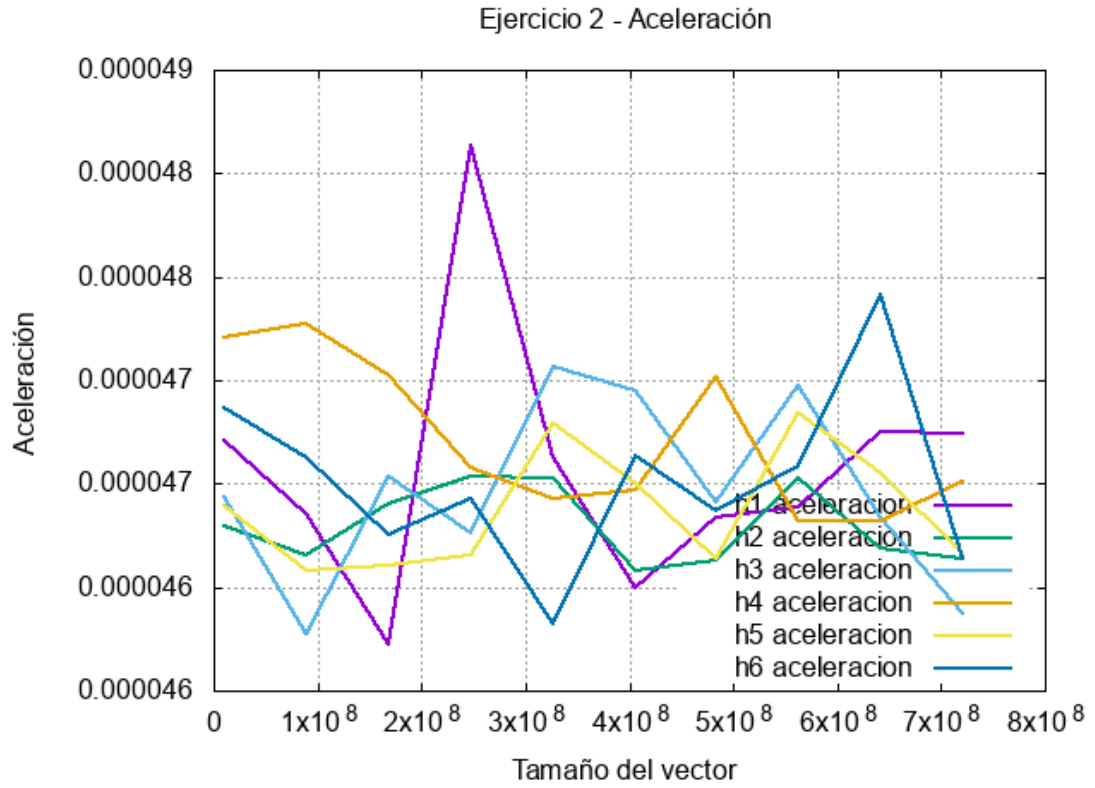
2.1. Observamos que dos de los resultados se aproximan mucho, mientras que otro es notablemente diferente. Ya que sabemos que el resultado del producto escalar es correcto, al no utilizar paralelización, tiene sentido que el algoritmo cuyo resultado se asemeja (PESCALAR_PAR2) sea el correcto y que PESCALAR_PAR1 sea el incorrecto.

2.2. ¿A qué se debe esta diferencia? Los programas PESCALAR_PAR1 y PESCALAR_PAR2 son exactamente iguales, excepto en que el primero utiliza la directiva #PRAGMA OMP PARALLEL FOR y el segundo utiliza #PRAGMA OMP PARALLEL FOR REDUCTION(+:SUM). El problema de los bucles paralelos reside en que cada hilo genera un resultado parcial, el cual, si no es recogido, provoca resultados erróneos al acceder a él los diferentes *threads* y modificar su valor. La directiva *reduction*, que utiliza el segundo programa, evita este problema al ir recogiendo los resultados parciales y añadiéndolos a la variable *sum*.

Tras esto, se va a realizar un análisis del rendimiento obtenido en el problema del producto escalar para diferentes números de hilos, en concreto, de 1 a 6, siendo 6 el número de *cores* de nuestro equipo. Tomando únicamente la ejecución de le versión en serie y de la versión paralela correcta, se va a analizar cómo varía la aceleración obtenida para vectores de diferentes tamaños.

Para ello, se ha desarrollado el script ejercicio2.sh, el cual se encarga de generar las mediciones y las gráficas correspondientes a los tiempos medios y a la aceleración. Los tamaños de los vectores varían entre 10.000.000 y 800.000.000 y se han realizado 50 repeticiones del experimento para tener medidas fidedignas. Estas son las dos gráficas obtenidas:


Ejercicio 2 - Tiempo de ejecución
Tiempo de ejecución en segundos
0.0026
0.0024
0.0022
0.002
0.0018
0.0016
0.0014
0.0012
0.001
0.0008
0.0006
0
1x10^8
2x10^8
3x10^8
4x10^8
5x10^8
6x10^8
7x10^8
8x10^8
Tamaño del vector
h1 serie
h2 serie
h3 serie
h4 serie
h5 serie
h6 serie
h1 paralelo
h2 paralelo
h3 paralelo
h4 paralelo
h5 paralelo
h6 paralelo


Ejercicio 2 - Aceleración
Aceleración
0.000049
0.000048
0.000048
0.000047
0.000047
0.000046
0.000046
0
1x10^8
2x10^8
3x10^8
4x10^8
5x10^8
6x10^8
7x10^8
8x10^8
Tamaño del vector
h1 aceleracion
h2 aceleracion
h3 aceleracion
h4 aceleracion
h5 aceleracion
h6 aceleracion

2.3. En términos del tamaño de los vectores, no siempre resulta ventajoso lanzar hilos. Debido a que la creación de hilos conlleva unos costes, cuando el tamaño de dichos vectores es reducido no resulta eficiente utilizar paralelización para su ejecución.

2.4. Como hemos dicho, cuando los vectores son pequeños, no compensa lanzar hilos. En dichos casos, el coste de crear y mantener estos hilos es superior al cálculo en serie.

2.5. En general, el rendimiento mejora al aumentar el número de hilos a trabajar.

2.6. Como ya hemos visto, no es rentable paralelizar una ejecución en estos dos casos: cuando el tamaño de los vectores no sea suficientemente grande y cuando no se sobrepase el número de hilos virtuales que soporta nuestro equipo.

2.7. Como podemos ver en la gráfica, a partir de un cierto tamaño (en concreto, 700.000.000) la aceleración tiende a estabilizarse. Esto significa que ya no existe tanta diferencia entre utilizar uno, dos o varios hilos. Sin embargo, notamos que la ejecución que utiliza 6 hilos (número que coincide con nuestros *cores*), si presenta un pico de aceleración en ese rango de tamaños, lo que indica que para dichos vectores resulta mucho más eficiente que las demás.

EJERCICIO 3

PARALELIZAR LA MULTIPLICACIÓN DE MATRICES

En este ejercicio, se va a estudiar el impacto de la paralelización de la multiplicación de matrices, utilizando el código desarrollado en la práctica anterior.

En este caso, el programa se compone de tres bucles anidados, por lo que a la hora de llevar a cabo la paralelización es razonable preguntarse qué bucle interesa paralelizar. Es por ello por lo que vamos a desarrollar tres versiones paralelas de la multiplicación de matrices, paralelizando los tres bucles existentes.

Probando varios tamaños, y buscando aquellos que tarden entre 10 y 60 segundos, se han seleccionado los tamaños 1400x1400 y 2200x2200 para poder estudiar dos casos diferentes.

El *Speedup* se ha calculado con la siguiente fórmula:

$$Speedup = \frac{Tiempo\ de\ ejecución\ en\ serie\ (s)}{Tiempo\ de\ ejecución\ en\ paralelo\ (s)}$$

Tiempos de ejecución en segundos para una matriz de **tamaño 1400**:

Versión/#hilos	1	2	3	4	5	6
Serie	10.6498					
Paralela-bucle1	10.7456	5.5471	4.1590	3.6066	3.4438	3.5110
Paralela-bucle2	10.2042	4.9508	3.3434	2.4918	2.0033	1.7520
Paralela-bucle3	10.2263	5.1411	3.3677	2.5428	2.1305	1.7336

Speedup tomando como referencia la versión serie para una matriz de **tamaño 1400**:

Versión/#hilos	1	2	3	4	5	6
Serie	1					
Paralela-bucle1	0.9910	1.9198	2.5606	2.9528	3.0924	3.03326
Paralela-bucle2	1.0436	2.1511	3.1853	4.2739	5.3161	6.0786
Paralela-bucle3	1.0414	2.0715	3.1623	4.1882	4.9987	6.1431

Tiempos de ejecución en segundos para una matriz de **tamaño 2200**:

Versión/#hilos	1	2	3	4	5	6
Serie	55.3402					
Paralela-bucle1	57.1460	24.0132	16.1053	13.4506	12.1633	11.5856
Paralela-bucle2	55.1152	27.9922	18.7901	14.2759	11.6855	9.9085
Paralela-bucle3	55.2916	27.5588	18.4467	14.0223	11.2793	9.4379

Speedup tomando como referencia la versión serie para una matriz de **tamaño 2200**:

Versión/#hilos	1	2	3	4	5	6
Serie	1					
Paralela-bucle1	0.9684	2.3045	3.4361	4.1143	4.5497	4.7766
Paralela-bucle2	1.0040	1.9769	2.9451	3.8764	4.7358	5.5851
Paralela-bucle3	1.0008	2.0080	3.0000	3.9465	4.9063	5.8636

Al observar estas tablas, notamos una gran mejora en el rendimiento al utilizar un mayor número de hilos. También es destacable que la paralelización reduce prácticamente a la mitad el tiempo de ejecución al pasar de 1 hilo a 2, reducción que también es importante al pasar de 2 a 3 hilos. Sin embargo, a partir de los 3 hilos la diferencia en tiempo de ejecución tan solo difiere en unos segundos.

3.1. La versión que peor rendimiento presenta es la primera, es decir, aquella que paraleliza el bucle más interno. Esto se debe a que, al tener 3 bucles anidados y paralelizar el interior, se tendrá que lanzar cada hilo al comienzo de cada iteración del bucle superior, para luego ser unidos al final de cada una de ellas. De este modo, se genera una cantidad innecesaria de hilos que lleven a cabo pequeñas tareas, por lo que al final el tiempo de la ejecución aumentará sin realmente presentar ningún beneficio.

3.2. En cuanto al mejor rendimiento, observamos que las versiones 2 y 3 presentan resultados muy similares. Esto se debe a que los bucles externos paralelos son más eficientes al generase una menor cantidad de hilos, lo que supone una reducción del tiempo de ejecución, al desplegarse dichos hilos al inicio de la ejecución y unirse al final. En concreto, la versión 3 debería tener resultados óptimos, al ser el bucle más exterior, pero se entiende que las medidas tomadas no resultan suficientes para apreciar dicha mejora.

Utilizando el script EJERCICIO3.SH, vamos a registrar los tiempos de ejecución para diferentes tamaños de matrices. Utilizaremos la versión paralela 3 con seis hilos, ya que hemos visto que es la más eficiente, y compararemos su rendimiento con la versión en serie.

EJERCICIO 4

EJEMPLO DE INTEGRACIÓN NUMÉRICA

El objetivo de este ejercicio es el de estudiar el valor aproximado del número π mediante una integración numérica. Vamos a ejecutar el programa de prueba PI_SERIE.C para analizar dichos valores. Esta es la salida del programa:

```
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_serie
Resultado pi: 3.141593
Tiempo 1.143129
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$
```

4.1. En esta versión del programa se utilizan 100.000.000 de rectángulos.

Ahora analizaremos el rendimiento de los programas pi_par1.c y pi_par4.c, los cuales son dos versiones paralelas del algoritmo anterior. Estas son sus salidas:

```
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par1
Numero de cores del equipo: 6
Resultado pi: 3.141593
Tiempo 0.348844
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par4
Numero de cores del equipo: 6
Resultado pi: 3.141593
Tiempo 0.249167
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$
```

Observamos que el tiempo que emplea la versión 1 es mayor que el de la versión 4.

4.2. La diferencia que podemos encontrar en el código de estas dos versiones reside en la forma de almacenar los resultados parciales del bucle for. La versión 1 almacena directamente en cada posición del array los resultados correspondientes, mientras que la versión 4 utiliza una variable privada para almacenar cada suma.

4.3. Como ya hemos visto en el ejercicio 2, es necesario recoger cada valor parcial de manera independiente, para que los resultados sean fidedignos. El motivo por el que la versión 1 tarda más se debe al *false sharing*, al tener cada hilo que actualizar el valor de la variable cada vez que otro o modifica, incrementándose así el tiempo de ejecución. Por otro lado, la versión 4 evita este efecto al utilizar una variable privada para almacenar los resultados correctos al salir del bucle, lo que hace que tarde menos tiempo en realizar su tarea.

Por último, vamos a analizar los programas PI_PAR2.C y PI_PAR3.C, los cuales incorporan dos modificaciones distintas sobre la versión 1 para intentar obtener un rendimiento similar a la versión 4. Estos son sus resultados:

```
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par2
Numero de cores del equipo: 6
Resultado pi: 3.141593
Tiempo 0.326857
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.238148
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$
```

Observamos que la versión 3 sí se asemeja al tiempo obtenido a la versión 4, mientras que la versión 2 tarda notablemente más tiempo

4.4. Ambas versiones tratan de asemejarse a la versión 4 de modo diferente. La versión 2 trata de resolver el problema del *false sharing* declarando la variable sum como *firstprivate*, para tratar de conservar su valor en la región compartida. Sin embargo, este efecto se sigue produciendo, por lo que el tiempo de ejecución sigue siendo alto. Por otro lado, la versión 3 sí logra combatir el *false sharing* mediante la reserva de memoria de un bloque entero de caché para almacenar cada hilo, lo cual acelera la ejecución.

Al cambiar el padding del ejercicio PI_PAR3 obtenemos los siguientes resultados:

```
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 1 elementos
Resultado pi: 3.141593
Tiempo 0.496049
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 2 elementos
Resultado pi: 3.141593
Tiempo 0.393081
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ make
```

```
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 4 elementos
Resultado pi: 3.141593
Tiempo 0.253147
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ make
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 6 elementos
Resultado pi: 3.141593
Tiempo 0.252174
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ make
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 7 elementos
Resultado pi: 3.141593
Tiempo 0.251704
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ make
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.229146
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ make
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 9 elementos
Resultado pi: 3.141593
Tiempo 0.223768
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ make
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 10 elementos
Resultado pi: 3.141593
Tiempo 0.234162
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ make
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$ ./pi_par3
Numero de cores del equipo: 6
Double size: 8 bytes
Cache line size: 64 bytes => padding: 12 elementos
Resultado pi: 3.141593
Tiempo 0.241879
e356372@15-19-15-19:~/UnidadH/ARQO/p4/materialP4$
```

4.5. Observamos que el rendimiento mejora al aumentar el *padding*. Eso sí, esto se cumple solo para los primeros valores, ya que vemos que en las últimas ejecuciones los resultados se estancan y no se nota mejora.

EJERCICIO 5

USO DE LA DIRECTIVA *CRITICAL* Y *REDUCTION*

En este ejercicio, se va a estudiar cómo resolver el problema de la falsa compartición mediante el uso de variables privadas y la creación de una sección crítica.

Al ejecutar las versiones 4 y 5 del programa PI_PAR, obtenemos los siguientes resultados:

```
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./pi_par4
Numero de cores del equipo: 6
Resultado pi: 3.141593
Tiempo 0.189426
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./pi_par5
Resultado pi: 2.930116
Tiempo 0.650911
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$
```

Se observa a primera vista que la versión 5 tarda notablemente más que la 4 y además que su resultado se aleja mucho de la realidad.

5.1. La directiva CRITICAL protege las variables, actuando como un *mutex*, de modo que los diferentes hilos no solapen sus valores y se generen resultados finales erróneos. Esta puede ser la causa de que versión 5, que es la que la utiliza, tarde más tiempo. Al ralentizar las zonas críticas la ejecución, debido a que no puede haber más de un hilo dentro de ellas, el programa tarda más tiempo. Por otro lado, el resultado incorrecto de la versión 5 se deba probablemente a que la variable i, que funciona de contador para el bucle for, no se ha declarado como privada, lo que provoca iteraciones con contadores incorrectos.

Al ejecutar las versiones 6 y 7 del programa PI_PAR, obtenemos los siguientes resultados:

```
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./pi_par6
Numero de cores del equipo: 6
Resultado pi: 3.141593
Tiempo 0.289831
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$ ./pi_par7
Resultado pi: 3.141593
Tiempo 0.191249
e356372@8B-31-19-31:~/UnidadH/ARQO/p4/materialP4$
```

En este caso, ambos resultados son correctos, pero el tiempo de ejecución de la versión 6 es significativamente más alto.

5.2. En la versión 6, se utiliza la directiva for, que paraleliza cada iteración del bucle. Esto no resulta muy eficiente, ya que se produce false sharing al utilizar un array para almacenar los resultados de cada hilo, además de que dichos resultados pueden ser incorrectos. Por otro lado, la versión 7 incluye la directiva reduction, la cual ya hemos estudiado en el ejercicio 2. Además de una significativa simplificación del código, esta directiva reduce los tiempos de ejecución y asegura resultados correctos.